W3School 正则表达式教程

wizardforcel

Published
with GitBook

目錄

介紹	0
正则表达式 - 简介	1
正则表达式 - 语法	2
正则表达式 - 元字符	3
正则表达式 - 运算符优先级	4
正则表达式 - 匹配规则	5
正则表达式 - 示例	6
免责声明	7

W3School 正则表达式教程

来源：正则表达式教程

整理：飞龙

正则表达式 - 简介

除非您以前使用过正则表达式，否则您可能不熟悉此术语。但是，毫无疑问，您已经使用过不涉及脚本的某些正则表达式概念。

例如，您很可能使用 ? 和 通配符来查找硬盘上的文件。通配符匹配文件名中的单个字符，而通配符匹配零个或多个字符。像 data?.dat 这样的模式将查找下列文件：

```
data1.dat
data2.dat
datax.dat
dataN.dat
```

使用 字符代替 ? 字符扩大了找到的文件的数量。*data*.dat 匹配下列所有文件：

```
data.dat
data1.dat
data2.dat
data12.dat
datax.dat
dataXYZ.dat
```

尽管这种搜索方法很有用，但它还是有限的。通过理解 * 通配符的工作原理，引入了正则表达式所依赖的概念，但正则表达式功能更强大，而且更加灵活。

正则表达式的使用，可以通过简单的办法来实现强大的功能。下面先给出一个简单的示例：

```
^.+@.+\\..+$
```

继续阅读本教程将让您也可以自由应用这样的代码。

为什么使用正则表达式？

典型的搜索和替换操作要求您提供与预期的搜索结果匹配的确切文本。虽然这种技术对于对静态文本执行简单搜索和替换任务可能已经足够了，但它缺乏灵活性，若采用这种方法搜索动态文本，即使不是不可能，至少也会变得很困难。

通过使用正则表达式，可以：

- 测试字符串内的模式。
  例如，可以测试输入字符串，以查看字符串内是否出现电话号码模式或信用卡号码模式。这称为数据验证。
- 替换文本。
  可以使用正则表达式来识别文档中的特定文本，完全删除该文本或者用其他文本替换

它。
- 基于模式匹配从字符串中提取子字符串。
  可以查找文档内或输入域内特定的文本。

例如，您可能需要搜索整个网站，删除过时的材料，以及替换某些 HTML 格式标记。在这种情况下，可以使用正则表达式来确定在每个文件中是否出现该材料或该 HTML 格式标记。此过程将受影响的文件列表缩小到包含需要删除或更改的材料的那些文件。然后可以使用正则表达式来删除过时的材料。最后，可以使用正则表达式来搜索和替换标记。

发展历史

正则表达式的"祖先"可以一直上溯至对人类神经系统如何工作的早期研究。Warren McCulloch 和 Walter Pitts 这两位神经生理学家研究出一种数学方式来描述这些神经网络。

1956 年, 一位叫 Stephen Kleene 的数学家在 McCulloch 和 Pitts 早期工作的基础上，发表了一篇标题为"神经网事件的表示法"的论文，引入了正则表达式的概念。正则表达式就是用来描述他称为"正则集的代数"的表达式，因此采用"正则表达式"这个术语。

随后，发现可以将这一工作应用于使用 Ken Thompson 的计算搜索算法的一些早期研究，Ken Thompson 是 Unix 的主要发明人。正则表达式的第一个实用应用程序就是 Unix 中的 qed 编辑器。

如他们所说，剩下的就是众所周知的历史了。从那时起直至现在正则表达式都是基于文本的编辑器和搜索工具中的一个重要部分。

应用领域

目前，正则表达式已经在很多软件中得到广泛的应用，包括 *nix（Linux, Unix等）、HP 等操作系统，PHP、C#、Java 等开发环境，以及很多的应用软件中，都可以看到正则表达式的影子。

C# 正则表达式

在我们的 C# 教程中，C# 正则表达式 这一章节专门介绍了有关 C# 正则表达式的知识。

Java 正则表达式

在我们的 Java 教程中，Java 正则表达式 这一章节专门介绍了有关 Java 正则表达式的知识。

JavaScript 正则表达式

在我们的 JavaScript 教程中，JavaScript RegExp 对象 这一章节专门介绍了有关 JavaScript 正则表达式的知识，同时我们还提供了完整的 JavaScript RegExp 对象参考手册。

Python 正则表达式

在我们的 Python 基础教程中，Python 正则表达式 这一章节专门介绍了有关 Python 正则表达式的知识。

Ruby 正则表达式

在我们的 Ruby 教程中，Ruby 正则表达式 这一章节专门介绍了有关 Ruby 正则表达式的知识。

正则表达式 - 语法

正则表达式(regular expression)描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。

- 列目录时， dir *.txt*或*ls* .txt中的*.txt*就不是一个正则表达式,因为这里与正则式的*的含义是不同的。
- 构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与运算符可以将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个的字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。

正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为"元字符"）组成的文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

普通字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符。这包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

非打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

字符	描述
\cx	匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

特殊字符

所谓特殊字符，就是一些有特殊含义的字符，如上面说的".*txt*"中的，简单的说就是表示任何字符串的意思。如果要查找文件名中有的文件，则需要对进行转义，即在其前加一个\。ls *.txt。

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符"转义"，即，将反斜杠字符 () 放在它们前面。下表列出了正则表达式中的特殊字符：

特别字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 $。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 (和)。
*	匹配前面的子表达式零次或多次。要匹配 字符，请使用 \。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 +。
.	匹配除换行符 \n之外的任何单字符。要匹配 .，请使用 \。
[	标记一个中括号表达式的开始。要匹配 [，请使用 [。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\' 匹配 "\"，而 '(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 {。
|	指明两项之间的一个选择。要匹配 |，请使用 \ |。

限定符

限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有*或+或?或{n}或{n,}或{n,m}共6种。

正则表达式的限定符有：

字符	描述
*	匹配前面的子表达式零次或多次。例如，zo 能匹配 *"z"* 以及 *"zoo"*。 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

由于章节编号在大的输入文档中会很可能超过九，所以您需要一种方式来处理两位或三位章节编号。限定符给您这种能力。下面的正则表达式匹配编号为任何位数的章节标题：

```
/Chapter [1-9][0-9]*/
```

请注意，限定符出现在范围表达式之后。因此，它应用于整个范围表达式，在本例中，只指定从 0 到 9 的数字（包括 0 和 9）。

这里不使用 + 限定符，因为在第二个位置或后面的位置不一定需要有一个数字。也不使用？字符，因为它将章节编号限制到只有两位数。您需要至少匹配 Chapter 和空格字符后面的一个数字。

如果您知道章节编号被限制为只有 99 章，可以使用下面的表达式来至少指定一位但至多两位数字。

```
/Chapter [0-9]{1,2}/
```

上面的表达式的缺点是，大于 99 的章节编号仍只匹配开头两位数字。另一个缺点是 Chapter 0 也将匹配。只匹配两位数字的更好的表达式如下：

```
/Chapter [1-9][0-9]?/
```

或

```
/Chapter [1-9][0-9]{0,1}/
```

***、+**和**?**限定符都是贪婪的，因为它们会尽可能多的匹配文字，只有在它们的后面加上一个**?**就可以实现非贪婪或最小匹配。

例如，您可能搜索 HTML 文档，以查找括在 H1 标记内的章节标题。该文本在您的文档中如下：

```
<H1>Chapter 1 – Introduction to Regular Expressions</H1>
```

下面的表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。

```
/<.*>/
```

如果您只需要匹配开始 H1 标记，下面的"非贪心"表达式只匹配 <H1>。

```
/<.*?>/
```

通过在 *、+ 或 ? 限定符之后放置 ?，该表达式从"贪心"表达式转换为"非贪心"表达式或者最小匹配。

定位符

定位符使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。

定位符用来描述字符串或单词的边界，^和$分别指字符串的开始与结束，\b描述单词的前或后边界，\B表示非单词边界。

正则表达式的限定符有：

字符	描述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个字边界，即字与空格间的位置。
\B	非字边界匹配。

注意：不能将限定符与定位点一起使用。由于在紧靠换行或者字边界的前面或后面不能有一个以上位置，因此不允许诸如 ^* 之类的表达式。

若要匹配一行文本开始处的文本，请在正则表达式的开始使用 ^ 字符。不要将 ^ 的这种用法与中括号表达式内的用法混淆。

若要匹配一行文本的结束处的文本，请在正则表达式的结束处使用 $ 字符。

若要在搜索章节标题时使用定位点，下面的正则表达式匹配一个章节标题，该标题只包含两个尾随数字，并且出现在行首：

```
/^Chapter [1-9][0-9]{0,1}/
```

真正的章节标题不仅出现行的开始处，而且它还是该行中仅有的文本。它即出现在行首又出现在同一行的结尾。下面的表达式能确保指定的匹配只匹配章节而不匹配交叉引用。通过创建只匹配一行文本的开始和结尾的正则表达式，就可做到这一点。

```
/^Chapter [1-9][0-9]{0,1}$/
```

匹配字边界稍有不同，但向正则表达式添加了很重要的能力。字边界是单词和空格之间的位置。非字边界是任何其他位置。下面的表达式匹配单词 Chapter 的开头三个字符，因为这三个字符出现字边界后面：

```
/\bCha/
```

\b 字符的位置是非常重要的。如果它位于要匹配的字符串的开始，它在单词的开始处查找匹配项。如果它位于字符串的结尾，它在单词的结尾处查找匹配项。例如，下面的表达式匹配单词 Chapter 中的字符串 ter，因为它出现在字边界的前面：

```
/ter\b/
```

下面的表达式匹配 Chapter 中的字符串 apt，但不匹配 aptitude 中的字符串 apt：

```
/\Bapt/
```

字符串 apt 出现在单词 Chapter 中的非字边界处，但出现在单词 aptitude 中的字边界处。对于 \B 非字边界运算符，位置并不重要，因为匹配不关心究竟是单词的开头还是结尾。

选择

用圆括号将所有选择项括起来，相邻的选择项之间用|分隔。但用圆括号会有一个副作用，是相关的匹配会被缓存，此时可用?:放在第一个选项前来消除这种副作用。

其中?:是非捕获元之一，还有两个非捕获元是?=和?!，这两个还有更多的含义，前者为正向预查，在任何开始匹配圆括号内的正则表达式模式的位置来匹配搜索字符串，后者为负向预查，在任何开始不匹配该正则表达式模式的位置来匹配搜索字符串。

反向引用

对一个正则表达式模式或部分模式两边添加圆括号将导致相关匹配存储到一个临时缓冲区中，所捕获的每个子匹配都按照在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始，最多可存储 99 个捕获的子表达式。每个缓冲区都可以使用 '\n' 访问，其中 n 为一个标识特定缓冲区的一位或两位十进制数。

可以使用非捕获元字符 '?:'、'?=' 或 '?!' 来重写捕获，忽略对相关匹配的保存。

反向引用的最简单的、最有用的应用之一，是提供查找文本中两个相同的相邻单词的匹配项的能力。以下面的句子为例：

```
Is is the cost of of gasoline going up up?
```

上面的句子很显然有多个重复的单词。如果能设计一种方法定位该句子，而不必查找每个单词的重复出现，那该有多好。下面的正则表达式使用单个子表达式来实现这一点：

```
/\b([a-z]+) \1\b/gi
```

捕获的表达式，正如 [a-z]+ 指定的，包括一个或多个字母。正则表达式的第二部分是对以前捕获的子匹配项的引用，即，单词的第二个匹配项正好由括号表达式匹配。\1 指定第一个子匹配项。字边界元字符确保只检测整个单词。否则，诸如"is issued"或"this is"之类的词组将不能正确地被此表达式识别。

正则表达式后面的全局标记 (g) 指示，将该表达式应用到输入字符串中能够查找到的尽可能多的匹配。表达式的结尾处的不区分大小写 (i) 标记指定不区分大小写。多行标记指定换行符的两边可能出现潜在的匹配。

反向引用还可以将通用资源指示符 (URI) 分解为其组件。假定您想将下面的 URI 分解为协议（ftp、http 等等）、域地址和页/路径：

```
http://www.w3cschool.cc:80/html/html-tutorial.html
```

下面的正则表达式提供该功能：

```
/(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)/
```

第一个括号子表达式捕获 Web 地址的协议部分。该子表达式匹配在冒号和两个正斜杠前面的任何单词。第二个括号子表达式捕获地址的域地址部分。子表达式匹配 / 或 : 之外的一个或多个字符。第三个括号子表达式捕获端口号（如果指定了的话）。该子表达式匹配冒号后面的零个或多个数字。只能重复一次该子表达式。最后，第四个括号子表达式捕获 Web 地址指定的路径和/或页信息。该子表达式能匹配不包括 # 或空格字符的任何字符序列。

将正则表达式应用到上面的 URI，各子匹配项包含下面的内容：

- 第一个括号子表达式包含"http"
- 第二个括号子表达式包含"www.w3cschool.cc"
- 第三个括号子表达式包含":80"
- 第四个括号子表达式包含"/html/html-tutorial.html"

正则表达式 - 元字符

下表包含了元字符的完整列表以及它们在正则表达式上下文中的行为：

字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个 向后引用、或一个八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 "\(" 则匹配 "("。
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。
$	匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。
*	匹配前面的子表达式零次或多次。例如，zo 能匹配 *"z"* 以及 *"zoo"*。 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
(pattern)	匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在VBScript 中使用 SubMatches 集合，在JScript 中则使用 $0…$9 属性。要匹配圆括号字符，请使用 '\(' 或 '\)'。
(?:pattern)	匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用 "或" 字符 (|) 来组合一个模式的各个部分是很有用。例如， 'industr(?:y|ies) 就是一个比 'industry|industries' 更简略的表达式。
	正向预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，'Windows

(? =pattern)	(?=95|98|NT|2000)' 能匹配 "Windows 2000" 中的 "Windows"， 但不能匹配 "Windows 3.1" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?!pattern)	负向预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如'Windows (?!95|98|NT|2000)' 能匹配 "Windows 3.1" 中的 "Windows"，但不能匹配 "Windows 2000" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
x|y	匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。
[xyz]	字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。
[^xyz]	负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\cx	匹配由 x 指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 [^0-9]。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'。
\W	匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'。
	匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数

\xn	字长。例如，'\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。
\num	匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。例如，'(.)\1' 匹配两个连续的相同字符。
\n	标识一个八进制转义值或一个向后引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。
\nm	标识一个八进制转义值或一个向后引用。如果 \nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果 \nm 之前至少有 n 个获取，则 n 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字 (0-7)，则 \nm 将匹配八进制转义值 nm。
\nml	如果 n 为八进制数字 (0-3)，且 m 和 l 均为八进制数字 (0-7)，则匹配八进制转义值 nml。
\un	匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (?)。

正则表达式 - 运算符优先级

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。

相同优先级的从左到右进行运算，不同优先级的运算先高后低。下表从最高到最低说明了各种正则表达式运算符的优先级顺序：

运算符	描述
\	转义符
(), (?:), (?=), []	圆括号和方括号
*, +, ?, {n}, {n,}, {n,m}	限定符
^, $, \任何元字符、任何字符	定位点和序列（即：位置和顺序）
|	替换，"或"操作 字符具有高于替换运算符的优先级，使得"m|food"匹配"m"或"food"。若要匹配"mood"或"food"，请使用括号创建子表达式，从而产生"(m|f)ood"。

正则表达式 - 匹配规则

基本模式匹配

一切从最基本的开始。模式，是正规表达式最基本的元素，它们是一组描述字符串特征的字符。模式可以很简单，由普通的字符串组成，也可以非常复杂，往往用特殊的字符表示一个范围内的字符、重复出现，或表示上下文。例如：

```
^once
```

这个模式包含一个特殊的字符^，表示该模式只匹配那些以once开头的字符串。例如该模式与字符串"once upon a time"匹配，与"There once was a man from NewYork"不匹配。正如如^符号表示开头一样，$符号用来匹配那些以给定模式结尾的字符串。

```
bucket$
```

这个模式与"Who kept all of this cash in a bucket"匹配，与"buckets"不匹配。字符^和$同时使用时，表示精确匹配（字符串与模式一样）。例如：

```
^bucket$
```

只匹配字符串"bucket"。如果一个模式不包括^和$，那么它与任何包含该模式的字符串匹配。例如：模式

```
once
```

与字符串

```
There once was a man from NewYork
Who kept all of his cash in a bucket.
```

是匹配的。

在该模式中的字母(o-n-c-e)是字面的字符，也就是说，他们表示该字母本身，数字也是一样的。其他一些稍微复杂的字符，如标点符号和白字符（空格、制表符等），要用到转义序列。所有的转义序列都用反斜杠()打头。制表符的转义序列是：\t。所以如果我们要检测一个字符串是否以制表符开头，可以用这个模式：

```
^\t
```

类似的，用\n表示"新行"，\r表示回车。其他的特殊符号，可以用在前面加上反斜杠，如反斜杠本身用\表示，句号.用.表示，以此类推。

字符簇

在INTERNET的程序中，正规表达式通常用来验证用户的输入。当用户提交一个FORM以后，要判断输入的电话号码、地址、EMAIL地址、信用卡号码等是否有效，用普通的基于字面的字符是不够的。

所以要用一种更自由的描述我们要的模式的办法，它就是字符簇。要建立一个表示所有元音字符的字符簇，就把所有的元音字符放在一个方括号里：

```
[AaEeIiOoUu]
```

这个模式与任何元音字符匹配，但只能表示一个字符。用连字号可以表示一个字符的范围，如：

```
[a-z] //匹配所有的小写字母
[A-Z] //匹配所有的大写字母
[a-zA-Z] //匹配所有的字母
[0-9] //匹配所有的数字
[0-9\.\-] //匹配所有的数字，句号和减号
[ \f\r\t\n] //匹配所有的白字符
```

同样的，这些也只表示一个字符，这是一个非常重要的。如果要匹配一个由一个小写字母和一位数字组成的字符串，比如"z2"、"t6"或"g7"，但不是"ab2"、"r2d3" 或"b52"的话，用这个模式：

```
^[a-z][0-9]$
```

尽管[a-z]代表26个字母的范围，但在这里它只能与第一个字符是小写字母的字符串匹配。

前面曾经提到^表示字符串的开头，但它还有另外一个含义。当在一组方括号里使用^是，它表示"非"或"排除"的意思，常常用来剔除某个字符。还用前面的例子，我们要求第一个字符不能是数字：

```
^[^0-9][0-9]$
```

这个模式与"&5"、"g7"及"-2"是匹配的，但与"12"、"66"是不匹配的。下面是几个排除特定字符的例子：

```
[^a-z] //除了小写字母以外的所有字符
[^\\\/\^] //除了(\)(/)(^)之外的所有字符
[^\"\'] //除了双引号(")和单引号(')之外的所有字符
```

特殊字符"." (点，句号)在正规表达式中用来表示除了"新行"之外的所有字符。所以模式"^.5$"与任何两个字符的、以数字5结尾和以其他非"新行"字符开头的字符串匹配。模式"."可以匹配任何字符串，除了空串和只包括一个"新行"的字符串。

PHP的正规表达式有一些内置的通用字符簇，列表如下：

字符簇	描述
[[:alpha:]]	任何字母
[[:digit:]]	任何数字
[[:alnum:]]	任何字母和数字
[[:space:]]	任何空白字符
[[:upper:]]	任何大写字母
[[:lower:]]	任何小写字母
[[:punct:]]	任何标点符号
[[:xdigit:]]	任何16进制的数字，相当于[0-9a-fA-F]

确定重复出现

到现在为止，你已经知道如何去匹配一个字母或数字，但更多的情况下，可能要匹配一个单词或一组数字。一个单词有若干个字母组成，一组数字有若干个单数组成。跟在字符或字符簇后面的花括号({})用来确定前面的内容的重复出现的次数。

字符簇	描述
^[a-zA-Z_]$	所有的字母和下划线
^[[:alpha:]]{3}$	所有的3个字母的单词
^a$	字母a
^a{4}$	aaaa
^a{2,4}$	aa,aaa或aaaa
^a{1,3}$	a,aa或aaa
^a{2,}$	包含多于两个a的字符串
^a{2,}	如：aardvark和aaab，但apple不行
a{2,}	如：baad和aaa，但Nantucket不行
\t{2}	两个制表符
.{2}	所有的两个字符

这些例子描述了花括号的三种不同的用法。一个数字，{x}的意思是"前面的字符或字符簇只出现x次"；一个数字加逗号，{x,}的意思是"前面的内容出现x或更多的次数"；两个用逗号分隔的数字，{x,y}表示"前面的内容至少出现x次，但不超过y次"。我们可以把模式扩展到更多的单词或数字：

```
^[a-zA-Z0-9_]{1,}$ //所有包含一个以上的字母、数字或下划线的字符串
^[0-9]{1,}$ //所有的正数
^\-{0,1}[0-9]{1,}$ //所有的整数
^\-{0,1}[0-9]{0,}\.{0,1}[0-9]{0,}$ //所有的小数
```

最后一个例子不太好理解，是吗？这么看吧：与所有以一个可选的负号(-{0,1})开头(^)、跟着0个或更多的数字([0-9]{0,})、和一个可选的小数点(.{0,1})再跟上0个或多个数字([0-9]{0,})，并且没有其他任何东西($)。下面你将知道能够使用的更为简单的方法。

特殊字符"?"与{0,1}是相等的，它们都代表着："0个或1个前面的内容"或"前面的内容是可选的"。所以刚才的例子可以简化为：

```
^\-?[0-9]{0,}\.?[0-9]{0,}$
```

特殊字符"*"与{0,}是相等的，它们都代表着"0个或多个前面的内容"。最后，字符"+"与 {1,}是相等的，表示"1个或多个前面的内容"，所以上面的4个例子可以写成：

```
^[a-zA-Z0-9_]+$ //所有包含一个以上的字母、数字或下划线的字符串
^[0-9]+$ //所有的正数
^\-?[0-9]+$ //所有的整数
^\-?[0-9]*\.?[0-9]*$ //所有的小数
```

当然这并不能从技术上降低正规表达式的复杂性，但可以使它们更容易阅读。

正则表达式 - 示例

简单表达式

正则表达式的最简单形式是在搜索字符串中匹配其本身的单个普通字符。例如，单字符模式，如 A，不论出现在搜索字符串中的何处，它总是匹配字母 A。下面是一些单字符正则表达式模式的示例：

```
/a/
/7/
/M/
```

可以将许多单字符组合起来以形成大的表达式。例如，以下正则表达式组合了单字符表达式：a、7 和 M。

```
/a7M/
```

请注意，没有串联运算符。只须在一个字符后面键入另一个字符。

字符匹配

句点 (.) 匹配字符串中的各种打印或非打印字符，只有一个字符例外。这个例外就是换行符 (\n)。下面的正则表达式匹配 aac、abc、acc、adc 等等，以及 a1c、a2c、a-c 和 a#c：

```
/a.c/
```

若要匹配包含文件名的字符串，而句点 (.) 是输入字符串的组成部分，请在正则表达式中的句点前面加反斜扛 () 字符。举例来说明，下面的正则表达式匹配 filename.ext：

```
/filename\.ext/
```

这些表达式只让您匹配"任何"单个字符。可能需要匹配列表中的特定字符组。例如，可能需要查找用数字表示的章节标题（Chapter 1、Chapter 2 等等）。

中括号表达式

若要创建匹配字符组的一个列表，请在方括号（[和]）内放置一个或更多单个字符。当字符括在中括号内时，该列表称为"中括号表达式"。与在任何别的位置一样，普通字符在中括号内表示其本身，即，它在输入文本中匹配一次其本身。大多数特殊字符在中括号表达式内出现时失去它们的意义。不过也有一些例外，如：

- 如果] 字符不是第一项，它结束一个列表。若要匹配列表中的] 字符，请将它放在第一位，紧跟在开始 [后面。
- \ 字符继续作为转义符。若要匹配 \ 字符，请使用 \\。

括在中括号表达式中的字符只匹配处于正则表达式中该位置的单个字符。以下正则表达式匹配 Chapter 1、Chapter 2、Chapter 3、Chapter 4 和 Chapter 5：

```
/Chapter [12345]/
```

请注意，单词 Chapter 和后面的空格的位置相对于中括号内的字符是固定的。中括号表达式指定的只是匹配紧跟在单词 Chapter 和空格后面的单个字符位置的字符集。这是第九个字符位置。

若要使用范围代替字符本身来表示匹配字符组，请使用连字符 (-) 将范围中的开始字符和结束字符分开。单个字符的字符值确定范围内的相对顺序。下面的正则表达式包含范围表达式，该范围表达式等效于上面显示的中括号中的列表。

```
/Chapter [1-5]/
```

当以这种方式指定范围时，开始值和结束值两者都包括在范围内。注意，还有一点很重要，按 Unicode 排序顺序，开始值必须在结束值的前面。

若要在中括号表达式中包括连字符，请采用下列方法之一：

- 用反斜扛将它转义：

  ```
  [\-]
  ```

- 将连字符放在中括号列表的开始或结尾。下面的表达式匹配所有小写字母和连字符：

  ```
  [-a-z]
  [a-z-]
  ```

- 创建一个范围，在该范围中，开始字符值小于连字符，而结束字符值等于或大于连字符。下面的两个正则表达式都满足这一要求：

  ```
  [!--]
  [!-~]
  ```

若要查找不在列表或范围内的所有字符，请将插入符号 (^) 放在列表的开头。如果插入字符出现在列表中的其他任何位置，则它匹配其本身。下面的正则表达式匹配编号大于 5 的章节标题：

```
/Chapter [^12345]/
```

在上面的示例中，表达式在第九个位置匹配 1、2、3、4 或 5 之外的任何数字字符。这样，例如，Chapter 7 就是一个匹配项，Chapter 9 也是一个匹配项。

上面的表达式可以使用连字符 (-) 来表示：

```
/Chapter [^1-5]/
```

中括号表达式的典型用途是指定任何大写或小写字母或任何数字的匹配。下面的表达式指定这样的匹配：

```
/[A-Za-z0-9]/
```

替换和分组

替换使用 | 字符来允许在两个或多个替换选项之间进行选择。例如，可以扩展章节标题正则表达式，以返回比章标题范围更广的匹配项。但是，这并不象您可能认为的那样简单。替换匹配 | 字符任一侧最大的表达式。

您可能认为，下面的表达式匹配出现在行首和行尾、后面跟一个或两个数字的 Chapter 或 Section：

```
/^Chapter|Section [1-9][0-9]{0,1}$/
```

很遗憾，上面的正则表达式要么匹配行首的单词 Chapter，要么匹配行尾的单词 Section 及跟在其后的任何数字。如果输入字符串是 Chapter 22，那么上面的表达式只匹配单词 Chapter。如果输入字符串是 Section 22，那么该表达式匹配 Section 22。

若要使正则表达式更易于控制，可以使用括号来限制替换的范围，即，确保它只应用于两个单词 Chapter 和 Section。但是，括号也用于创建子表达式，并可能捕获它们以供以后使用，这一点在有关反向引用的那一节讲述。通过在上面的正则表达式的适当位置添加括号，就可以使该正则表达式匹配 Chapter 1 或 Section 3。

下面的正则表达式使用括号来组合 Chapter 和 Section，以便表达式正确地起作用：

```
/^(Chapter|Section) [1-9][0-9]{0,1}$/
```

尽管这些表达式正常工作，但 Chapter|Section 周围的括号还将捕获两个匹配字中的任一个供以后使用。由于在上面的表达式中只有一组括号，因此，只有一个被捕获的"子匹配项"。

在上面的示例中，您只需要使用括号来组合单词 Chapter 和 Section 之间的选择。若要防止匹配被保存以备将来使用，请在括号内正则表达式模式之前放置 ?:。下面的修改提供相同的能力而不保存子匹配项：

```
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/
```

除 ?: 元字符外，两个其他非捕获元字符创建被称为"预测先行"匹配的某些内容。正向预测先行使用 ?= 指定，它匹配处于括号中匹配正则表达式模式的起始点的搜索字符串。反向预测先行使用 ?! 指定，它匹配处于与正则表达式模式不匹配的字符串的起始点的搜索字符串。

例如，假设您有一个文档，该文档包含指向 Windows 3.1、Windows 95、Windows 98 和 Windows NT 的引用。再进一步假设，您需要更新该文档，将指向 Windows 95、Windows 98 和 Windows NT 的所有引用更改为 Windows 2000。下面的正则表达式（这是一个正向预测先行的示例）匹配 Windows 95、Windows 98 和 Windows NT：

```
/Windows(?=95 |98 |NT )/
```

找到一处匹配后，紧接着就在匹配的文本（不包括预测先行中的字符）之后搜索下一处匹配。例如，如果上面的表达式匹配 Windows 98，将在 Windows 之后而不是在 98 之后继续搜索。

其他示例

下面列出一些正则表达式示例：

<table>
<tr><th>正则表达式</th><th>描述</th></tr>
<tr><td>/\b([a-z]+) \1\b/gi</td><td>一个单词连续出现的位置。</td></tr>
<tr><td>/(\w+):\/\/([^/:]+)(:\d)?([^#])/</td><td>将一个URL解析为协议、域、端口及相对路径。</td></tr>
<tr><td>/^(?:Chapter|Section)[1-9][0-9]{0,1}$/</td><td>定位章节的位置。</td></tr>
<tr><td>/[-a-z]/</td><td>A至z共26个字母再加一个-号。</td></tr>
<tr><td>/ter\b/</td><td>可匹配chapter，而不能匹配terminal。</td></tr>
<tr><td>/\Bapt/</td><td>可匹配chapter，而不能匹配aptitude。</td></tr>
<tr><td>/Windows(?=95 |98 |NT)/</td><td>可匹配Windows95或Windows98或WindowsNT，当找到一个匹配后，从Windows后面开始进行下一次的检索匹配。</td></tr>
<tr><td>/^\s*$/</td><td>匹配空行。</td></tr>
<tr><td>/\d{2}-\d{5}/</td><td>验证由两位数字、一个连字符再加 5 位数字组成的 ID 号。</td></tr>
<tr><td>/<\s(\S+)(\s[^>])?>[\s\S]<\s\/\1\s*>/</td><td>匹配 HTML 标记。</td></tr>
</table>

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。